Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 1183–1205. © Cambridge University Press 2011
doi:10.1017/S0960129511000181

Non-termination and secure information flow[†]

GEOFFREY SMITH and RAFAEL ALPÍZAR

School of Computing and Information Sciences,
Florida International University, Miami, FL 33199 U.S.A.
Email: {smithg;ralpizar}@cis.fiu.edu

Received 2 June 2008; revised 28 September 2010

In secure information flow analysis, the classic Denning restrictions allow a program's termination to be affected by the values of its H variables, resulting in potential information leaks. In an effort to quantify such leaks, in this paper we study a simple imperative language with random assignments. As a thought experiment, we propose a 'stripping' operation on programs, which eliminates all 'high computation', and prove the fundamental property that stripping cannot decrease the probability of any low outcome. To prove this property, we first introduce a new notion of *fast probabilistic simulation* on Markov chains and show that it implies a key reachability property. Viewing the stripping function as a binary relation, we then prove that stripping is a fast simulation. As an application, we prove that, under the Denning restrictions, well-typed probabilistic programs are guaranteed to satisfy an approximate probabilistic non-interference property, provided that their probability of non-termination is small.

1. Introduction

The aim of secure information flow analysis is to prevent untrusted programs from leaking the sensitive information that they manipulate. If we classify variables as H (high) or L (low), our goal is to prevent information in H variables from flowing into L variables (more generally, we may want to use a richer *lattice* of security levels). The seminal work in this area was Denning and Denning (1977), which proposed what we call the *Denning restrictions*:

— An expression is classified as H if it contains any H variables, otherwise it is classified as L.
— To prevent *explicit flows*, an H expression cannot be assigned to an L variable.
— To prevent *implicit flows*, an **if** or **while** command whose guard is H may not make *any* assignments to L variables.

Much later, Volpano *et al.* (1996) showed the soundness of the Denning restrictions by formulating them as a type system and proving that they ensure a *non-interference* property. Much work has followed – see Sabelfeld and Myers (2003) for a survey up to 2003.

[†] This work was partially supported by the National Science Foundation under grants HRD-0317692 and CNS-0831114. An early version of this work appeared as Smith and Alpízar (2007).

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

```
t ←? {0,1};
if t = 0 then (
    while h = 1 do done;
    l := 0
)
else (
    while h = 0 do done;
    l := 1
)
```

Fig. 1. A random assignment program.

Non-interference says, roughly, that the final values of L variables are independent of the initial values of H variables. It is formalised using the concept of *low equivalence* of memories.

Definition 1.1. Memories μ and ν are *low equivalent*, written $\mu \sim_L \nu$, if μ and ν agree on the values of all L variables.

The idea is that if initial memories μ and ν are low equivalent, then running c under μ should be (in some sense) equivalent to running c under ν as far as L variables are concerned. But the possibility of non-termination causes complications because the Denning restrictions allow programs whose termination behaviour depends on the values of H variables. For example, the program **while** $h = 1$ **do done** (where h is an H variable) might terminate under μ and loop under ν.

One reaction to the termination issue is to say that further restrictions are needed. A number of studies have proposed forbidding H variables in the guards of **while** loops (see, for example, Volpano and Smith (1997)) or forbidding assignments to L variables that sequentially follow commands whose termination depends on H variables (see, for example, Smith (2006)). But such additional restrictions may be overly stringent in practice, making it difficult to write useful programs. For this reason, practical secure information-flow languages like Jif (Myers *et al.* 2006) have chosen *not* to impose extra restrictions to control termination leaks.

In this paper, therefore, we study the behaviour of potentially non-terminating programs typed under the Denning restrictions alone. In order that we can make *quantitative* statements about the effects of non-termination, we consider a *probabilistic* language. In such a language, we would like to achieve *probabilistic non-interference* (Volpano and Smith 1999), which asserts that the probability distribution on the final values of L variables is independent of the initial values of H variables.

For example, consider the program in Figure 1 (in the code, $t \xleftarrow{?} \{0,1\}$ is a *random assignment*, assigning either 0 or 1 to t, each with probability $1/2$, and **done** is like **skip**). Assuming that h is H and t and l are L, this program satisfies the Denning restrictions and is well typed under the typing rules we will present in Section 2. However, if $h = 0$, this program terminates with $l = 0$ with probability $1/2$ and fails to terminate with probability $1/2$. And if $h = 1$, it terminates with $l = 1$ with probability $1/2$ and fails to terminate with probability $1/2$. Thus this program does not satisfy probabilistic non-interference.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

```
t ←? {0, 1};
if t = 0 then
    l := 0
else
    l := 1
```

Fig. 2. Stripped version of the program.

What goes wrong? Intuitively, the program is 'trying' to set l to either 0 or 1, each with probability $1/2$, regardless of the value of h. But the **while** loops, whose termination depends on the value of h, sometimes prevent assignments to l from being reached. As a result, the probabilities of certain final values of l are lowered because the paths that would have led to them become infinite loops. This suggests, perhaps, that if a well-typed program's probability of non-termination is small, it will 'almost' satisfy probabilistic non-interference.

To make these intuitions precise, we consider a thought experiment. For any well-typed program c, we define a 'stripped' version of c, which we denote by $\lfloor c \rfloor$. In $\lfloor c \rfloor$, all purely 'high computation' is removed; more precisely, we eliminate any subcommands that make no assignments to L variables. For example, the stripped version of the program in Figure 1 is shown in Figure 2

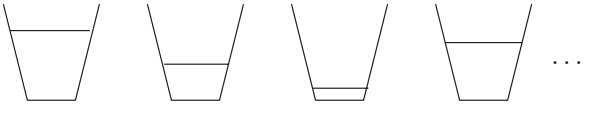
Note that stripping is for us purely a *thought experiment* to help us *understand* a program's behaviour, and is not something that we would actually use in an implementation.

The major technical effort of the current paper is to prove a precise relationship between the behaviour of a well-typed program c and that of its stripped version $\lfloor c \rfloor$. We will show that the only effect of the stripping operation is to boost the probabilities of certain L outcomes by lowering the probability of non-termination. For example, consider the program in Figure 1 when $h = 0$. Stripping boosts the probability of terminating with $l = 1$ from 0 up to $1/2$ by lowering the probability of non-termination from $1/2$ down to 0; it leaves the probability of terminating with $l = 0$ unchanged at $1/2$.

More precisely, we will prove in Theorem 4.3 that the probability that c terminates with certain values for its L variables is always less than or equal to the corresponding probability for $\lfloor c \rfloor$.

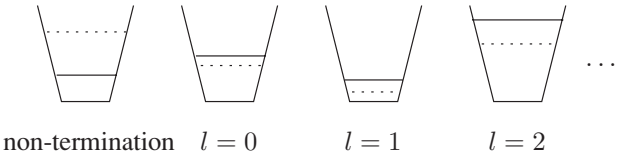
To visualise this theorem, imagine that the result of running c is shown as a sequence of buckets, one for each possible final value of c's L variables. Also, we have a bucket to represent non-termination. The probability of each outcome is indicated by the amount of water in each bucket; there is a total of one gallon of water distributed between all the buckets.

Suppose c's buckets have the following contents:



https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

Theorem 4.3 then tells us that $\lfloor c \rfloor$'s buckets are simply derived by pouring some of the water from c's non-termination bucket into some of the other buckets:



Our Theorem 4.3 was claimed, without proof, as Theorem 3.6 of our earlier paper Smith and Alpízar (2006). Furthermore, we claimed there that the theorem could be proved using *strong probabilistic simulation* as defined in Jonsson and Larsen (1991). However, that claim is not correct since strong simulation turns out to be too restrictive for this purpose. For this reason, we introduce a new probabilistic simulation, which we call *fast simulation*, and show that it implies a key reachability property. Viewing the stripping function as a binary relation, we then prove in Theorem 4.2 the crucial fact that *stripping is a fast simulation*, which implies Theorem 4.3.

Theorem 4.3 gives us a way of bounding the effect of non-termination. For example, if c's non-termination bucket is empty, then $\lfloor c \rfloor$'s buckets are identical to c's because there is no water to pour. More generally, we prove in Theorem 5.2 that if a well-typed program c fails to terminate with probability at most p, then c's deviation from probabilistic non-interference is at most $2p$.

We see this paper as making two main contributions. First, it gives a quantitative account of information flows caused by non-termination in programs that satisfy the Denning restrictions alone; this is important for understanding more precisely what is guaranteed in languages that allow termination channels. Second, this paper makes a technical contribution by introducing a new notion of simulation, called *fast simulation*, and applying it to the area of secure information flow; as far as we are aware, probabilistic simulation (unlike probabilistic *bisimulation*) has not previously been used in the study of secure information flow.

The rest of this paper is organised as follows. In Section 2, we review the simple imperative language with random assignment that we will study. In Section 3, we explore the theory of probabilistic simulation in the abstract setting of Markov chains, developing a variant that we call *fast simulation*. In Section 4, we define the stripping operation $\lfloor \cdot \rfloor$ formally and use the theory of fast simulation to prove Theorem 4.3, which gives a fundamental relationship between the behaviour of a well-typed command and its stripped version. In Section 5, we develop several applications of these results. Finally, Section 6 presents related work and our conclusions.

2. A random assignment language

In this section, we review the syntax, semantics and type system for our simple imperative language with random assignment. The language syntax is defined in Figure 3. In the syntax, metavariables x, y, z range over identifiers and n over integer literals. Integers are the only values; we use 0 for false and non-zero for true.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

$$
\begin{array}{lrcl}
(\textit{phrases}) & p & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
(\textit{commands}) & c & ::= & \textbf{done} \mid x := e \mid x \overset{?}{\leftarrow} \mathcal{D} \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c \mid c_1; c_2
\end{array}
$$

Fig. 3. Language syntax.

A novelty of our language is that we have replaced the usual **skip** command with a **done** command; **done** can be used in much the same way as **skip** and (as will be seen below) it is also used to represent a terminated command in a configuration.

The command $x \overset{?}{\leftarrow} \mathscr{D}$ is a random assignment; here $\mathscr{D}$ ranges over some set of probability distributions on the integers. In examples, we use notation like $x \overset{?}{\leftarrow} \{0, 1, 2\}$ to denote a random assignment command that assigns 0, 1 or 2 to x, each with equal probability. Note that our language allows random *assignments* but not random *expressions*, which lets us use a simpler semantics.

A program c is executed under a *memory* μ, which maps identifiers to integers. We assume that expressions are total and evaluated atomically, and we write $\mu(e)$ to denote the value of expression e in memory μ. In our semantics, a *configuration* is a pair (c, μ) where c is a command and μ is a memory. Note that *terminal* configurations are written as $(\textbf{done}, \mu)$ in our semantics. Note also that the more common semantic approach is to have both non-terminal configurations (c, μ) and terminal configurations μ, but this tends to lead to a proliferation of cases in proofs, so it is more pleasant to have only configurations of the form (c, μ).

Because of the random assignment command, the standard transition relation on configurations needs to be extended with probabilities, and we write $(c, \mu) \xrightarrow{p} (c', \mu')$ to indicate that the probability of going from configuration (c, μ) to configuration (c', μ') is p. The semantic rules are given in Figure 4. They define a *Markov chain* (Feller 1968) on the set of configurations. Notice that terminal configurations $(\textbf{done}, \mu)$ are *absorbing states* in the Markov chain.

Next we describe our type system, which simply enforces the Denning restrictions. The types it uses are

$$
\begin{array}{lrcl}
(\textit{data types}) & \tau & ::= & L \mid H \\
(\textit{phrase types}) & \rho & ::= & \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd}.
\end{array}
$$

As usual, τ *var* is the type of variables that store information of level τ, while τ *cmd* is the type of commands that assign only to variables of level τ or higher, which implies that command types obey a *contravariant* subtyping rule. Typing judgments have the form $\Gamma \vdash p : \rho$, where Γ is an *identifier typing* that maps each variable to a type of the form τ *var*. (We generally assume a single fixed Γ throughout, so in our discussions we will seldom mention Γ.) The typing and subtyping rules are given in Figure 5. The rules are the same as those in Volpano *et al.* (1996), except for the new rules $done_t$, which types **done** as a high command, and $random_t$ for random assignment. Rule $random_t$ says that a

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

$$(done_s)\quad (\mathbf{done},\mu)\xrightarrow{1}(\mathbf{done},\mu)$$

$$(update_s)\quad \frac{x\in dom(\mu)}{(x:=e,\mu)\xrightarrow{1}(\mathbf{done},\mu[x:=\mu(e)])}$$

$$(random_s)\quad \frac{x\in dom(\mu)\ \ \mathcal{D}(v)>0}{(x\xleftarrow{?}\mathcal{D},\mu)\xrightarrow{\mathcal{D}(v)}(\mathbf{done},\mu[x:=v])}$$

$$(if_s)\quad \frac{\mu(e)\neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2,\mu)\xrightarrow{1}(c_1,\mu)}\qquad \frac{\mu(e)=0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2,\mu)\xrightarrow{1}(c_2,\mu)}$$

$$(while_s)\quad \frac{\mu(e)=0}{(\mathbf{while}\ e\ \mathbf{do}\ c,\mu)\xrightarrow{1}(\mathbf{done},\mu)}\qquad \frac{\mu(e)\neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c,\mu)\xrightarrow{1}(c;\mathbf{while}\ e\ \mathbf{do}\ c,\mu)}$$

$$(compose_s)\quad \frac{(c_1,\mu)\xrightarrow{p}(\mathbf{done},\mu')}{(c_1;c_2,\mu)\xrightarrow{p}(c_2,\mu')}\qquad \frac{(c_1,\mu)\xrightarrow{p}(c_1',\mu')\ \ c_1'\neq\mathbf{done}}{(c_1;c_2,\mu)\xrightarrow{p}(c_1';c_2,\mu')}$$

Fig. 4. Structural operational semantics.

random assignment can be done to any variable x, but if x has type $\tau\ var$, the assignment gets type $\tau\ cmd$; this type is used prevent improper implicit flows.

Under this type system, we have the usual Simple Security, Confinement and Subject Reduction properties.

Lemma 2.1 (Simple security). If $\Gamma\vdash e:\tau$, then e contains only variables of level τ or lower.

Proof. The proof is by induction on the structure of e. □

Lemma 2.2 (Confinement). If $\Gamma\vdash c:\tau\ cmd$, then c assigns only to variables of level τ or higher.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

$$(rval_t)\quad \frac{\Gamma(x) = \tau\ var}{\Gamma \vdash x : \tau}$$

$$(int_t)\quad \Gamma \vdash n : L$$

$$(plus_t)\quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$$

$$(done_t)\quad \Gamma \vdash \textbf{done} : H\ cmd$$

$$(update_t)\quad \frac{\Gamma(x) = \tau\ var \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\ cmd}$$

$$(random_t)\quad \frac{\Gamma(x) = \tau\ var}{\Gamma \vdash x \stackrel{?}{\leftarrow} \mathcal{D} : \tau\ cmd}$$

$$(if_t)\quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau\ cmd \quad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd}$$

$$(while_t)\quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau\ cmd}{\Gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd}$$

$$(compose_t)\quad \frac{\Gamma \vdash c_1 : \tau\ cmd \quad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash c_1; c_2 : \tau\ cmd}$$

$$(base)\quad L \subseteq H$$

$$(cmd)\quad \frac{\tau \subseteq \tau'}{\tau'\ cmd \subseteq \tau\ cmd}$$

$$(reflex)\quad \rho \subseteq \rho$$

$$(trans)\quad \frac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$(subsump)\quad \frac{\Gamma \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

Fig. 5. Typing and subtyping rules.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

Proof. The proof is by induction on the structure of c. □

Lemma 2.3 (Subject reduction). If $\Gamma \vdash c : \tau\ cmd$ and $(c, \mu) \xrightarrow{p} (c', \mu')$ for some $p > 0$, then $\Gamma \vdash c' : \tau\ cmd$.

Proof. The proof is by induction on the structure of c. □

However, as discussed in the *Introduction*, well-typed programs need not satisfy probabilistic non-interference because changes to H variables can result in infinite loops that block subsequent assignments to L variables, thus affecting the probabilities of different L outcomes. Before undertaking a deeper study of the behaviour of well-typed programs, we will need to develop some useful results in the theory of probabilistic simulation.

3. Probabilistic simulation

In this section, we discuss the theory of probabilistic bisimulation and simulation in an abstract setting.

A (discrete-time) *Markov chain* (Feller 1968) is a pair $(S, \mathbf{P})$ where:

— S is a countable set of states; and
— $\mathbf{P} : S \times S \to [0, 1]$ is a probability matrix satisfying $\sum_{t \in S} \mathbf{P}(s, t) = 1$ for all $s \in S$.

If $\mathbf{P}(s, t) > 0$, we say that t is a *successor* of s. Also, for $T \subseteq S$, we write $\mathbf{P}(s, T)$ to denote $\sum_{t \in T} \mathbf{P}(s, t)$, which is the probability of going in one step from s to a state in T.

A classic equivalence relation on Markov chains is *probabilistic bisimulation*, which is due to Kemeny and Snell (1960) and Larsen and Skou (1991).

Definition 3.1. Let R be an equivalence relation on S. R is a *strong bisimulation* if whenever sRt we have $\mathbf{P}(s, T) = \mathbf{P}(t, T)$ for every equivalence class T of R.

Both strong and weak versions of bisimulation have been applied fruitfully in secure information flow analysis: for example, in Gray (1990), Sabelfeld and Sands (2000) and Smith (2006). The basic idea is that a secure program should behave (in some sense) 'indistinguishably' when run under two low-equivalent initial memories. This indistinguishability can be formalised as a bisimulation.

In this paper, we instead apply non-symmetric *probabilistic simulation* relations, which were explored earlier in Jonsson and Larsen (1991) and Baier *et al.* (2005). Roughly speaking, a binary relation R is a strong simulation if whenever sRt and s has a successor s', then t has a 'matching' successor t' that simulates s' (that is, $s' R t'$). But in the probabilistic setting, we must also make sure that the probabilities are preserved. Suppose, for example, that $\mathbf{P}(s, s') = p$. Then t must be able to match the same probability. However, t need not have a single successor t' such that $\mathbf{P}(t, t') = p$. Instead, t may have several successors, each simulating s', and it is enough that the total probability is p. However, in doing this simulation we must not 'double count' t's probabilities – for example, if s goes to s' with probability $1/3$ and t goes to t' with probability $1/2$, then if we use t' to match the move to s' we must remember that $1/3$ of t''s probability is 'used up', leaving just $1/6$ to be used in matching other moves of s. These considerations lead to what is called a *weight*

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

function Δ to specify how the probabilities are matched up, as in the following definition (which is adapted from Baier *et al.* (2005, Definition 16)).

Definition 3.2. Let R be a binary relation on S. R is a *strong simulation* if, whenever sRt, there exists a function $\Delta : S \times S \to [0,1]$ such that:

(1) $\Delta(s',t') > 0$ implies that $s'Rt'$.
(2) $\mathbf{P}(s,s') = \sum_{u \in S} \Delta(s',u)$ for all $s' \in S$.
(3) $\mathbf{P}(t,t') = \sum_{u \in S} \Delta(u,t')$ for all $t' \in S$.

However, for our exploration of the stripping operation $\lfloor \cdot \rfloor$ in Section 4, it turns out that strong simulation is not quite what we want because it does not allow the simulating state t to run 'faster' than the simulated state s. The issue is that s could make 'insignificant' moves to states that are already simulated by t, and in this case t should not need to make a matching move. Such 'insignificant' moves are allowed by the more flexible notion of *weak simulation* in Baier *et al.* (2005, Definition 34). However, their definition also allows t to make 'insignificant' moves, which are not appropriate for us here since we want t to run at least as fast as s. So here we develop a restricted kind of weak simulation, which we call a *fast simulation*.

Suppose sRt. We partition the successors of s into two sets U and V. The states in V represent 'insignificant' moves, and we require that t itself simulates each of them. The states in U represent 'significant' moves, and we require that t be able to match such moves with some weight function. Intuitively, then, t matches s's moves either by doing nothing or by moving. Formally, we have the following definition.

Definition 3.3. Let R be a binary relation on S. R is a *fast simulation* if, whenever sRt, the successors of s can be partitioned into two sets U and V such that:

(1) vRt for every $v \in V$; and
(2) letting $K = \sum_{u \in U} \mathbf{P}(s,u)$, if $K > 0$, there exists a function $\Delta : S \times S \to [0,1]$ such that:
   (a) $\Delta(u,w) > 0$ implies that uRw.
   (b) $\mathbf{P}(s,u)/K = \sum_{w \in S} \Delta(u,w)$ for all $u \in U$.
   (c) $\mathbf{P}(t,w) = \sum_{u \in U} \Delta(u,w)$ for all $w \in S$.

Notice that in condition 2(b), $\mathbf{P}(s,u)/K$ is the *conditional probability* of going from s to u, given that s goes to U. (The reason for using a conditional probability here may not be intuitively clear; in fact, the best justification for this definition is its utility in the proof of the key Theorem 3.2 below.)

Example 3.1. Consider the Markov chain in Figure 6, where $S = \{s,t,s',t',w\}$. Define R by sRt, $s'Rt'$, together with uRu for every $u \in S$. (In other words, R is the *reflexive closure* of $\{(s,t),(s',t')\}$.) Then we can show that R is a fast simulation:

— For pairs of the form xRx, we can always satisfy the requirements of Definition 3.3 by choosing U to be the set of successors of x and V to be $\varnothing$. Then $K = 1$, and for each $u \in U$, we can choose $\Delta(u,u) = \mathbf{P}(x,u)$. It is straightforward to verify that these choices satisfy conditions 1, 2(a), 2(b) and 2(c).

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

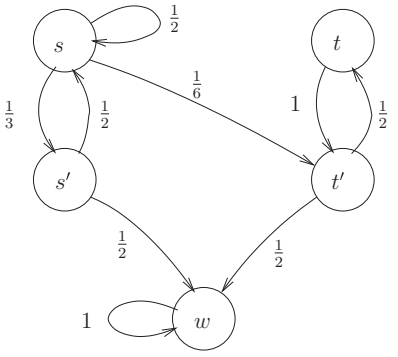


Fig. 6. An example Markov chain.

— For sRt, we can choose $U = \{s', t'\}$ and $V = \{s\}$, which makes $K = 1/2$, and we can choose $\Delta(s', t') = 2/3$ and $\Delta(t', t') = 1/3$.
— Finally, for $s'Rt'$ we can choose $U = \{s, w\}$ and $V = \varnothing$, which makes $K = 1$, and we can choose $\Delta(s, t) = 1/2$ and $\Delta(w, w) = 1/2$.

Note that every strong simulation is also a fast simulation, since a strong simulation is just a fast simulation in which all the V sets are empty. Furthermore, every fast simulation is also a weak simulation as defined in Baier *et al.* (2005, Definition 34).

We will now develop the key properties of fast simulation.

Definition 3.4. Let R be a binary relation on S. A set T of states is *upwards closed with respect to* R if, whenever $s \in T$ and sRs', we also have $s' \in T$.

With $s \in S$, n a natural number and $T \subseteq S$, we write $\Pr(s, n, T)$ to denote the probability of reaching a state in T from s in at most n steps. Following Baier *et al.* (2005), we can calculate $\Pr(s, n, T)$ using a recurrence:

$$\Pr(s, n, T) = \begin{cases} 1 & \text{if } s \in T \\ \sum_{s' \in S} \mathbf{P}(s, s') \Pr(s', n-1, T) & \text{if } n > 0 \text{ and } s \notin T \\ 0 & \text{if } n = 0 \text{ and } s \notin T. \end{cases}$$

Note that $\Pr(s, n, T)$ increases monotonically with n.

Lemma 3.1. $\Pr(s, n, T) \leqslant \Pr(s, n+1, T)$, for all n, s and T.

Proof. The proof is by induction on n. For the basis, if $s \notin T$, then

$$\Pr(s, 0, T) = 0 \leqslant \Pr(s, 1, T),$$

and if $s \in T$, then

$$\Pr(s, 0, T) = 1 = \Pr(s, 1, T).$$

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

For the inductive step, we assume that for some $k \geqslant 0$ we have

$$\Pr(s, k, T) \leqslant \Pr(s, k+1, T)$$

for all s and T. We must show that

$$\Pr(s, k+1, T) \leqslant \Pr(s, k+2, T).$$

First note that if $s \in T$, we have

$$\Pr(s, k+1, T) = 1 = \Pr(s, k+2, T).$$

We still need to consider the case when $s \notin T$. In this case we have

$$\begin{aligned}\Pr(s, k+2, T) &= \sum_{s' \in S} \mathbf{P}(s, s') \Pr(s', k+1, T) \\ &\geqslant \sum_{s' \in S} \mathbf{P}(s, s') \Pr(s', k, T) \qquad \text{(by induction)} \\ &= \Pr(s, k+1, T).\end{aligned}$$

□

We now proceed to the key theorem about fast simulation: the proof is similar to the proof of Baier *et al.* (2005, Theorem 54), though that theorem refers to *strong simulation.*

Theorem 3.2. If R is a fast simulation, T is upwards closed with respect to R, and $s_1 R s_2$, then $\Pr(s_1, n, T) \leqslant \Pr(s_2, n, T)$ for every n.

Proof. The proof is by induction on n. For the basis, note that if $s_1 \notin T$, then $\Pr(s_1, 0, T) = 0 \leqslant \Pr(s_2, 0, T)$, and if $s_1 \in T$, then $s_2 \in T$, since $s_1 R s_2$ and T is upwards closed. So we have $\Pr(s_1, 0, T) = 1 = \Pr(s_2, 0, T)$.

For the inductive step, first note that if $s_1 \in T$, then, as above, we have

$$\Pr(s_1, k+1, T) = 1 = \Pr(s_2, k+1, T),$$

and if $s_2 \in T$, we have

$$\Pr(s_1, k+1, T) \leqslant 1 = \Pr(s_2, k+1, T).$$

We still need to consider the case when $s_1 \notin T$ and $s_2 \notin T$. In this case we have

$$\begin{aligned}\Pr(s_1, k+1, T) &= \sum_{s \in S} \mathbf{P}(s_1, s) \Pr(s, k, T) \\ &= \sum_{v \in V} \mathbf{P}(s_1, v) \Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u) \Pr(u, k, T)\end{aligned}$$

where U and V are as specified in Definition 3.3 (note that the rearrangement is valid because the series are absolutely convergent).

Now, for every $v \in V$, since $v R s_2$, we get by induction that $\Pr(v, k, T) \leqslant \Pr(s_2, k, T)$. Also, letting $K = \sum_{u \in U} \mathbf{P}(s_1, u)$, we note that $\sum_{v \in V} \mathbf{P}(s_1, v) = (1 - K)$. Hence we have

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

$$\begin{aligned}\sum_{v\in V}\mathbf{P}(s_1,v)\Pr(v,k,T) &\leqslant \sum_{v\in V}\mathbf{P}(s_1,v)\Pr(s_2,k,T)\\ &= \left(\sum_{v\in V}\mathbf{P}(s_1,v)\right)\Pr(s_2,k,T)\\ &= (1-K)\Pr(s_2,k,T)\\ &\leqslant (1-K)\Pr(s_2,k+1,T).\end{aligned}$$

Note that if $K=0$, then $U=\varnothing$, which implies that

$$\begin{aligned}\Pr(s_1,k+1,T) &= \sum_{v\in V}\mathbf{P}(s_1,v)\Pr(v,k,T)+\sum_{u\in U}\mathbf{P}(s_1,u)\Pr(u,k,T)\\ &\leqslant (1-0)\Pr(s_2,k+1,T)+0\\ &= \Pr(s_2,k+1,T).\end{aligned}$$

We are left with the case when $K>0$. Now, by Definition 3.3, there exists a function Δ satisfying conditions 2(a), 2(b) and 2(c). Hence we have

$$\begin{aligned}\sum_{u\in U}\mathbf{P}(s_1,u)\Pr(u,k,T) &= \sum_{u\in U}\left(K\sum_{w\in S}\Delta(u,w)\right)\Pr(u,k,T)\\ &= K\sum_{u\in U}\left(\sum_{w\in S}\Delta(u,w)\Pr(u,k,T)\right)\\ &\leqslant K\sum_{u\in U}\left(\sum_{w\in S}\Delta(u,w)\Pr(w,k,T)\right)\\ &\qquad\qquad \text{(by induction, since } \Delta(u,w)>0 \text{ implies } uRw)\\ &= K\sum_{w\in S}\left(\sum_{u\in U}\Delta(u,w)\Pr(w,k,T)\right)\\ &= K\sum_{w\in S}\left(\sum_{u\in U}\Delta(u,w)\right)\Pr(w,k,T)\\ &= K\sum_{w\in S}\mathbf{P}(s_2,w)\Pr(w,k,T)\\ &= K\Pr(s_2,k+1,T).\end{aligned}$$

Finally, we have

$$\begin{aligned}\Pr(s_1,k+1,T) &= \sum_{v\in V}\mathbf{P}(s_1,v)\Pr(v,k,T)+\sum_{u\in U}\mathbf{P}(s_1,u)\Pr(u,k,T)\\ &\leqslant (1-K)\Pr(s_2,k+1,T)+K\Pr(s_2,k+1,T)\\ &= \Pr(s_2,k+1,T).\end{aligned}$$

□

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

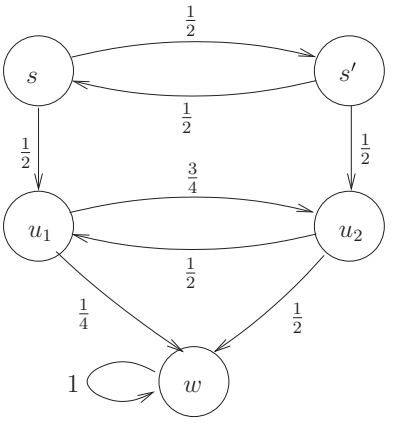


Fig. 7. Another example Markov chain.

We can illustrate Theorem 3.2 by considering the Markov chain in Example 3.1 above and its fast simulation R. Because $s'Rt'$ and $\{w\}$ is upwards closed with respect to R, Theorem 3.2 tells us that $\Pr(s', n, \{w\}) \leqslant \Pr(t', n, \{w\})$ for every n. Direct calculation confirms these inequalities for $n \leqslant 5$:

n	$\Pr(s', n, \{w\})$	$\Pr(t', n, \{w\})$
0	0	0
1	$\frac{1}{2}$	$\frac{1}{2}$
2	$\frac{1}{2}$	$\frac{1}{2}$
3	$\frac{5}{8}$	$\frac{3}{4}$
4	$\frac{11}{16}$	$\frac{3}{4}$
5	$\frac{73}{96}$	$\frac{7}{8}$

Note that the *universal relation* $R_U = S \times S$ is trivially a fast simulation. But under R_U the only upwards-closed sets are $\varnothing$ and S itself, which means that Theorem 3.2 is uninteresting in that case.

Note that Theorem 3.2 also holds if R is a strong simulation, since every strong simulation is also a fast simulation. Interestingly, Theorem 3.2 *fails* if R is a weak simulation as defined in Baier *et al.* (2005, Definition 34). Example 3.2 provides a counterexample.

Example 3.2. Consider the Markov chain in Figure 7, which is taken from Baier *et al.* (2005, page 197). If R is the reflexive closure of $\{(s, s'), (u_1, u_2)\}$, then R is a weak simulation and $\{w\}$ is upwards closed with respect to R, but $\Pr(s, 3, \{w\}) = 7/16$ and $\Pr(s', 3, \{w\}) = 6/16$. Notice that in this case R is not a fast simulation.

Now that we have developed fast simulations on abstract Markov chains, we are almost ready to apply them to our study of the stripping operation $\lfloor \cdot \rfloor$. However, it turns out that when we study the behaviour of $\lfloor \cdot \rfloor$ we will not need the great flexibility allowed by

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

fast simulation as defined in Definition 3.3. We therefore introduce a restricted kind of fast simulation as follows.

Definition 3.5. A binary relation R on S is a *simple fast simulation* if, whenever sRt, either

(1) for every successor s' of s, we have $s'Rt$; or
(2) there is a bijection δ from the successors of s to the successors of t such that for every successor s' of s, we have $\mathbf{P}(s,s') = \mathbf{P}(t,\delta(s'))$ and $s'R\delta(s')$.

Compared with the fast simulation of Definition 3.3, case 1 here corresponds to the situation where all of the moves from s are 'inessential', allowing us to take $U = \varnothing$. And case 2 corresponds to the situation where all of the moves from s are 'essential', allowing us to take $V = \varnothing$, and, moreover, we can use a very simple weight function that pairs up the successors of s and the successors of t in a one-to-one manner.

We will now show that every simple fast simulation is indeed a fast simulation.

Theorem 3.3. Every simple fast simulation R is a fast simulation.

Proof. Suppose R is a simple fast simulation and sRt.

If case 1 holds, then we satisfy Definition 3.3 by letting V be the set of successors of s and letting U be $\varnothing$. Note in this case that $K = 0$, so condition 2 of Definition 3.3 is satisfied vacuously.

If case 2 holds, then we let U be the set of successors of s and let V be $\varnothing$. Now condition 1 of Definition 3.3 is satisfied vacuously. To satisfy condition 2, note that $K = 1$ and define $\Delta(s', \delta(s')) = \mathbf{P}(s,s') = \mathbf{P}(t,\delta(s'))$. (All other values of Δ are 0.) It is straightforward to verify that Δ satisfies conditions 2(a), 2(b) and 2(c). □

4. Stripping and its properties

In this section, we formally define the stripping operation on well-typed commands in our random assignment language and use our results about fast simulation to prove a fundamental result about the relationship between the behaviours of c and $\lfloor c \rfloor$. Intuitively, $\lfloor c \rfloor$ eliminates all subcommands of c that contain no assignments to L variables; it is easy to see that this is the same as eliminating subcommands of type $H\ cmd$. More precisely, we have the following definition, an early version of which appeared in Smith and Alpízar (2006).

Definition 4.1. Let c be a well-typed command. We define $\lfloor c \rfloor = \mathbf{done}$ if c has type $H\ cmd$, and otherwise define $\lfloor c \rfloor$ by:

— $\lfloor x := e \rfloor = x := e$
— $\lfloor x \overset{?}{\leftarrow} \mathscr{D} \rfloor = x \overset{?}{\leftarrow} \mathscr{D}$
— $\lfloor \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rfloor = \mathbf{if}\ e\ \mathbf{then}\ \lfloor c_1 \rfloor\ \mathbf{else}\ \lfloor c_2 \rfloor$
— $\lfloor \mathbf{while}\ e\ \mathbf{do}\ c \rfloor = \mathbf{while}\ e\ \mathbf{do}\ \lfloor c \rfloor$
— $\lfloor c_1; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \text{if } c_1 : H\ cmd \\ \lfloor c_1 \rfloor & \text{if } c_2 : H\ cmd \\ \lfloor c_1 \rfloor; \lfloor c_2 \rfloor & \text{otherwise.} \end{cases}$

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ and extend $\lfloor \cdot \rfloor$ to well-typed configurations through $\lfloor (c, \mu) \rfloor = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

Note that stripping as defined in Smith and Alpízar (2007) replaced subcommands of type $H\ cmd$ with **skip**; but in our new definition here, stripping aggressively *eliminates* such subcommands in sequential compositions and replaces them with **done** in the other cases. Note that $\lfloor \mu \rfloor \sim_L \mu$. We now have a simple lemma.

Lemma 4.1. For any command c, we have $\lfloor c \rfloor$ contains only L variables.

Proof. The proof is by induction on the structure of c. If c has type $H\ cmd$, then $\lfloor c \rfloor =$ **done**, which (vacuously) contains only L variables. If c does not have type $H\ cmd$, we consider each form of c in turn:

— c is $x := e$, so $\lfloor c \rfloor = x := e$.
Since c does not have type $H\ cmd$, then by rule $update_t$ we must have that x is an L variable and $e : L$, which implies by Simple Security that e contains only L variables.
— c is $x \overset{?}{\leftarrow} \mathscr{D}$.
This case is similar to the previous one.
— c is **if** e **then** c_1 **else** c_2, so $\lfloor c \rfloor =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$.
By rule if_t, we have $e : L$, which implies by Simple Security that e contains only L variables. Also, by induction, $\lfloor c_1 \rfloor$ and $\lfloor c_2 \rfloor$ contain only L variables.
— c is **while** e **do** c_1, so $\lfloor c \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$.
By rule $while_t$, $e : L$, which implies by Simple Security that e contains only L variables. Also, by induction, $\lfloor c_1 \rfloor$ contains only L variables.
— c is $c_1 ; c_2$.
If $c_1 : H\ cmd$, then $\lfloor c \rfloor = \lfloor c_2 \rfloor$, which by induction contains only L variables.
The other subcases are similar. □

We now specialise fast simulation from arbitrary Markov chains to the particular Markov chain of well-typed configurations (c, μ) of our random assignment language. In addition, we impose the requirement that the simulating configuration's memory must be low equivalent to the simulated configuration's memory.

Definition 4.2. A binary relation R on well-typed configurations is a *fast low simulation* if R is a fast simulation such that whenever $(c_1, \mu_1) R (c_2, \mu_2)$, we have $\mu_1 \sim_L \mu_2$.

Recall that in set theory any function is also a relation. Hence, our stripping function can also be viewed as a relation. When we view it that way, we use the notation $\lfloor \cdot \rfloor$ to denote the stripping relation, and write $(c_1, \mu_1) \lfloor \cdot \rfloor (c_2, \mu_2)$ to denote $\lfloor (c_1, \mu_1) \rfloor = (c_2, \mu_2)$.

We are now ready for our main theorem, which is that the relation $\lfloor \cdot \rfloor$ is a fast low simulation. But first we give an example that illustrates how the behaviours of (c, μ) and $\lfloor (c, \mu) \rfloor$ can differ. Consider the program

$$\begin{array}{l} \textbf{while}\ h\ \textbf{do}\ h \overset{?}{\leftarrow} \{0,1\}; \\ l := 1 \end{array}$$

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

and its stripped version

$$l := 1.$$

Notice that under memory $\{h = 1, l = 0\}$ the original program can run for an arbitrary number of steps; it has infinitely many terminating traces, whose probabilities sum to 1. In contrast, the stripped program always terminates in exactly one step.

Theorem 4.2. $\lfloor\cdot\rfloor$ is a fast low simulation.

Proof. First note that if $(c, \mu)\lfloor\cdot\rfloor(d, v)$, we have $v = \lfloor\mu\rfloor$, which implies $\mu \sim_L v$. More substantially, we must show that $\lfloor\cdot\rfloor$ is a fast simulation. By Theorem 3.3, it suffices to show the stronger result that $\lfloor\cdot\rfloor$ is a simple fast simulation. Note that the stronger result is *easier* to prove because it gives us a stronger induction hypothesis to use in the case of $c_1; c_2$. (This is an example of what George Pólya called the 'Inventor's Paradox'.)

Now suppose that $(c, \mu)\lfloor\cdot\rfloor(d, v)$, which means that $d = \lfloor c\rfloor$ and $v = \lfloor\mu\rfloor$. We must show that either condition 1 or 2 of Definition 3.5 holds, which we will do by induction on the structure of c.

First, if c has type $H\ cmd$, then $d = \mathbf{done}$. Now consider the (possibly infinite) set of successors of (c, μ):

$$\begin{aligned}(c, \mu) &\xrightarrow{p_1} (c_1, \mu_1)\\ (c, \mu) &\xrightarrow{p_2} (c_2, \mu_2)\\ (c, \mu) &\xrightarrow{p_3} (c_3, \mu_3)\\ &\dots\end{aligned}$$

For every i, by Subject Reduction, we have $c_i : H\ cmd$, and by Confinement, we have $\mu_i \sim_L \mu$. Hence we have $(c_i, \mu_i)\lfloor\cdot\rfloor(d, v)$. Thus we can see that condition 1 of Definition 3.5 is satisfied.

Next, if c does not have type $H\ cmd$, we consider each of the possible forms of c in turn:

(1) $c = x := e$.

So we have $d = c$. By $update_t$, we have $x : L\ var$ and $e : L$. So by Simple Security and the fact that $v = \lfloor\mu\rfloor$, we have $\mu(e) = v(e)$, which implies that $v[x := v(e)] = \lfloor\mu[x := \mu(e)]\rfloor$. Hence the move $(c, \mu)\xrightarrow{1}(\mathbf{done}, \mu[x := \mu(e)])$ is matched by the move $(d, v)\xrightarrow{1}(\mathbf{done}, v[x := v(e)])$. Formally, condition 2 of Definition 3.5 is satisfied by choosing

$$\delta((\mathbf{done}, \mu[x := \mu(e)])) = (\mathbf{done}, v[x := v(e)]).$$

(2) $c = x \xleftarrow{?} \mathscr{D}$.

This case is similar to case (1). Again $d = c$ and each move $(c, \mu)\xrightarrow{\mathscr{D}(v)}(\mathbf{done}, \mu[x := v])$ is matched by $(d, v)\xrightarrow{\mathscr{D}(v)}(\mathbf{done}, v[x := v])$. Formally, for each v such that $\mathscr{D}(v) > 0$, we choose

$$\delta((\mathbf{done}, \mu[x := v])) = (\mathbf{done}, v[x := v]).$$

(3) $c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

So we have $d = \textbf{if } e \textbf{ then } \lfloor c_1 \rfloor \textbf{ else } \lfloor c_2 \rfloor$. By if_t, we have $e : L$ and by Simple Security, $\mu(e) = v(e)$. So if $\mu(e) \neq 0$, then $(c, \mu) \xrightarrow{1} (c_1, \mu)$ is matched by $(d, v) \xrightarrow{1} (\lfloor c_1 \rfloor, v)$. Formally, we choose

$$\delta((c_1, \mu)) = (\lfloor c_1 \rfloor, v).$$

The case when $\mu(e) = 0$ is similar.

(4) $c = \textbf{while } e \textbf{ do } c_1$.

So we have $d = \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor$. By $while_t$, we have $e : L$ and c_1 does not have type $H\ cmd$. By Simple Security, $\mu(e) = v(e)$. So if $\mu(e) \neq 0$, the move

$$(c, \mu) \xrightarrow{1} (c_1 ; \textbf{while } e \textbf{ do } c_1, \mu)$$

is matched by

$$(d, v) \xrightarrow{1} (\lfloor c_1 \rfloor ; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor, v)$$

– notice that because c_1 does not have type $H\ cmd$, we have

$$\lfloor c_1 ; \textbf{while } e \textbf{ do } c_1 \rfloor = \lfloor c_1 \rfloor ; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor.$$

Formally, we choose

$$\delta((c_1 ; \textbf{while } e \textbf{ do } c_1, \mu)) = (\lfloor c_1 \rfloor ; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor, v).$$

The case when $\mu(e) = 0$ is similar.

(5) $c = c_1 ; c_2$.

First note that under rule $compose_s$, any move from $(c_1 ; c_2, \mu)$ results from a move

$$(c_1, \mu) \xrightarrow{p} (c_1', \mu')$$

where c_1' may or may not be **done**.

If $c_1' = \textbf{done}$, the move is

$$(c_1 ; c_2, \mu) \xrightarrow{p} (c_2, \mu'),$$

and if $c_1' \neq \textbf{done}$, the move is

$$(c_1 ; c_2, \mu) \xrightarrow{p} (c_1' ; c_2, \mu').$$

Now we consider subcases, depending on the types of c_1 and c_2:

(a) If c_1 has type $H\ cmd$, then $d = \lfloor c_2 \rfloor$. By Subject Reduction, we have $c_1' : H\ cmd$, and by Confinement we have $\mu' \sim_L \mu$. Hence (d, v) can match either of the possible moves from $(c_1 ; c_2, \mu)$ by doing nothing, since we have both $(c_2, \mu')\lfloor \cdot \rfloor (d, v)$ and $(c_1' ; c_2, \mu')\lfloor \cdot \rfloor (d, v)$. So condition 1 of Definition 3.5 is satisfied[†].

(b) If neither c_1 nor c_2 has type $H\ cmd$, then $d = \lfloor c_1 \rfloor ; \lfloor c_2 \rfloor$. We define $d_1 = \lfloor c_1 \rfloor$ and $d_2 = \lfloor c_2 \rfloor$. Now, by induction, (d_1, v) can match the moves from (c_1, μ) so

[†] Note that this is the critical case that prevents us from showing that $\lfloor \cdot \rfloor$ is a strong simulation. For example, if c is $h := 2; l := 3$ and μ is $\{h = 0, l = 1\}$, then d is $l := 3$ and v is $\{l = 1\}$. In this case the move $(c, \mu) \xrightarrow{1} (l := 3, \{h = 2, l = 1\})$ cannot be matched by the move $(d, v) \xrightarrow{1} (\textbf{done}, \{l = 3\})$, since the resulting memories are not low equivalent. Instead (d, v) must match the move by doing nothing.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

that either condition 1 or 2 of Definition 3.5 is satisfied. We consider these two possibilities in turn:

— If condition 1 is satisfied, we always have $(c_1', \mu')\lfloor\cdot\rfloor(d_1, v)$. Now observe that d_1 cannot be **done**, since $d_1 = \lfloor c_1 \rfloor$ and c_1 does not have type $H\ cmd$. This implies that c_1' cannot have type $H\ cmd$ and, in particular, that c_1' is not **done**. Hence each move from c must be $(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')$. Moreover, we have $(c_1'; c_2, \mu')\lfloor\cdot\rfloor(d_1; d_2, v)$. Hence condition 1 is satisfied for $(c_1; c_2, \mu)$ and $(d_1; d_2, v)$.

— If condition 2 is satisfied, there is a bijection δ from the successors of (c_1, μ) to the successors of (d_1, v). If we let (d_1', v') denote $\delta((c_1', \mu'))$, then under condition 2 we have $(d_1, v) \xrightarrow{p} (d_1', v')$ and $(c_1', \mu')\lfloor\cdot\rfloor(d_1', v')$.

Now consider c_1'.

If $c_1' =$ **done**, the move from $(c_1; c_2, \mu)$ is $(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')$. In this case we also have $d_1' =$ **done**, so we have the matching move $(d_1; d_2, v) \xrightarrow{p} (d_2, v')$.

If $c_1' \neq$ **done**, the move from $(c_1; c_2, \mu)$ is $(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')$. This move is matched by $(d_1; d_2, v) \xrightarrow{p} (d_1'; d_2, v')$, but only if c_1' does not have type $H\ cmd$. This is because if $c_1' : H\ cmd$, then $\lfloor c_1'; c_2 \rfloor = \lfloor c_2 \rfloor = d_2 \neq d_1; d_2$, but in this case we have $d_1' =$ **done**, which means that we actually have the matching move $(d_1; d_2, v) \xrightarrow{p} (d_2, v')$[†].

(c) Finally, if c_1 does not have type $H\ cmd$ and $c_2 : H\ cmd$, the argument is essentially the same as in case (b). □

We can now prove that if (c, μ) is well typed and can terminate within at most n steps with its L variables having certain values, then $\lfloor (c, \mu) \rfloor$ can do the same, with probability at least as great (this property was claimed, without proof, as Smith and Alpízar (2006, Theorem 3.6)).

We will say that a *low memory property* Φ is any property that depends only on the values of L variables. For example, if x and y are L variables, then '$x = 5$ and y is even' is a low memory property.

Theorem 4.3. Let c be well typed and Φ be a low memory property. For any n, the probability that (c, μ) terminates within n steps in a final memory satisfying Φ is less than or equal to the corresponding probability for $\lfloor (c, \mu) \rfloor$.

Proof. By Theorem 4.2, $\lfloor\cdot\rfloor$ is a fast simulation. Let $T = \{(\textbf{done}, v) \mid v \text{ satisfies } \Phi\}$. It is easy to see that T is upwards closed with respect to $\lfloor\cdot\rfloor$ since if $(\textbf{done}, v_1) \in T$ and $(\textbf{done}, v_1)\lfloor\cdot\rfloor(\textbf{done}, v_2)$, then v_1 satisfies Φ and $v_1 \sim_L v_2$, which implies that v_2 also satisfies Φ. Therefore we can apply Theorem 3.2 to deduce that $\Pr((c, \mu), n, T) \leqslant \Pr(\lfloor (c, \mu) \rfloor, n, T)$ for every n. □

[†] An example illustrating this scenario is when c is (**if** 0 **then** $l := 1$ **else** $h := 2$); $l := 3$. This goes in one step to $h := 2; l := 3$, which strips to $l := 3$. In this case,

$$\lfloor c \rfloor = (\textbf{if } 0 \textbf{ then } l := 1 \textbf{ else done}); l := 3,$$

which goes *in one step* to $l := 3$.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

Note that we can extend Theorem 4.3 to the case of *eventually* terminating in T, since the probability of eventually terminating in T is just $\lim_{n\to\infty} \Pr((c,\mu), n, T)$.

5. Applications

Theorem 4.3 gives us the ability to quantify how the behaviour of a well-typed program c can deviate from its stripped version $\lfloor c \rfloor$. But it also gives us a way to quantify how the behaviour of c under memory μ can deviate from its behaviour under ν, assuming that μ and ν are low equivalent. The reason is that, by Lemma 4.1, $\lfloor c \rfloor$ contains only L variables, which means that its behaviour under μ must be *identical* to its behaviour under ν. Hence we can build a 'bridge' between (c,μ) and (c,ν):

$$(c,\mu) \xleftrightarrow{\text{Theorem 4.3}} (\lfloor c \rfloor, \lfloor \mu \rfloor) \equiv (\lfloor c \rfloor, \lfloor \nu \rfloor) \xleftrightarrow{\text{Theorem 4.3}} (c,\nu).$$

In this section, we develop several applications of these ideas.

First, suppose c is well typed and *probabilistically total*, which means that it halts with probability 1 from all initial memories. Then (c,μ)'s non-termination bucket is empty, which implies by Theorem 4.3 that (c,μ)'s buckets are identical to $(\lfloor c \rfloor, \lfloor \mu \rfloor)$'s buckets. Similarly, (c,ν)'s buckets are identical to $(\lfloor c \rfloor, \lfloor \nu \rfloor)$'s buckets. Hence, (c,μ)'s buckets are identical to (c,ν)'s buckets. So we have proved the following corollary.

Corollary 5.1. If c is well typed and probabilistically total, then c satisfies probabilistic non-interference.

This same result was proved in a different way as Smith and Alpízar (2006, Corollary 3.5) – the proof there used a weak probabilistic bisimulation.

More interestingly, we can now prove an *approximate probabilistic non-interference* result for well-typed programs whose probability of non-termination is bounded.

Corollary 5.2. Suppose c is well typed and fails to terminate from any initial memory with probability at most p. If μ and ν are low equivalent, then the deviation between the distributions of L outcomes under μ and under ν is at most $2p$.

Proof. Since (c,μ)'s non-termination bucket contains at most p units of water, the sum of the absolute value of the differences between the L outcome buckets of (c,μ) and of $(\lfloor c \rfloor, \lfloor \mu \rfloor)$ is at most p, and similarly for (c,ν). Hence the sum of the absolute value of the differences between the L outcome buckets of (c,μ) and of (c,ν) is at most $2p$. □

Notice that the program in Figure 1 achieves the upper bound of this corollary. From any initial memory, this program fails to terminate with probability at most $1/2$, so here $p = 1/2$. When $h = 0$, it terminates with $l = 0$ with probability $1/2$ and terminates with $l = 1$ with probability 0. When $h = 1$, it terminates with $l = 0$ with probability 0 and terminates with $l = 1$ with probability $1/2$. Hence the deviation between the two distributions of L outcomes is $|1/2 - 0| + |0 - 1/2| = 1$, which is $2p$. In general, applying Corollary 5.2 usefully requires a good bound p on the probability of non-termination; of course such bounds may be hard to obtain.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

As an application of the approximate non-interference result, notice that an adversary $\mathscr{A}$, given the final values of c's L variables, might try to distinguish between initial memories μ and ν through statistical hypothesis testing. Assuming that the probability p of non-termination is small, then the approximate non-interference property gives us a way to bound $\mathscr{A}$'s ability to do this. Similar ideas are considered (in the context of a process algebra) in Di Pierro *et al.* (2002).

Finally, we stress that Theorem 4.3 is critical to all of the main results of our earlier paper Smith and Alpízar (2006). For instance, Theorem 4.1 of that paper is a reduction proof that shows that it is sound to give type L to the encryption of an H expression, under a suitably chosen and protected symmetric key, provided the encryption scheme is IND-CPA secure[†]. (Note that such a rule violates the Denning restrictions, which would classify $\mathscr{E}(h)$ as H.) We define a *leaking adversary* $\mathscr{B}$ to be a program in the language of Smith and Alpízar (2006) that tries to leak a randomly initialised one-bit H variable h into an L variable l. If $\mathscr{B}$ succeeds with probability q, then its *leaking advantage* is $2q-1$. (It is defined in this way because $\mathscr{B}$ can trivially succeed with probability $1/2$.) Smith and Alpízar (2006, Theorem 4.1) shows that if a leaking adversary $\mathscr{B}$ is well typed under the Denning restrictions together with the above rule for typing encryptions, then there exists an IND-CPA adversary $\mathscr{A}$ such that:

— $\mathscr{A}$ is about as efficient as $\mathscr{B}$; and
— $\mathscr{A}$'s IND-CPA advantage is at least half of $\mathscr{B}$'s leaking advantage.

The reduction works roughly as follows. First, $\mathscr{A}$ runs $\mathscr{B}$ with a randomly chosen value of h. Then, whenever $\mathscr{B}$ calls its encryption primitive $\mathscr{E}(e)$, $\mathscr{A}$ passes the pair $(0^n, e)$ to its LR oracle and returns the result to $\mathscr{B}$. When $\mathscr{B}$ terminates, $\mathscr{A}$ guesses that $b=1$ if $\mathscr{B}$ was successful in leaking h and guesses that $b=0$ otherwise. Now, if $b=1$, then $\mathscr{B}$ is run *faithfully*, and if $b=0$, then $\mathscr{B}$ is run simply as a random assignment program (since its encryptions $\mathscr{E}(e)$ just return $\mathscr{E}_K(0^n)$, which is a random number that has nothing to do with e). Subtly, this introduces the possibility of non-termination when $b=0$. Hence, to prove that $\mathscr{A}$'s IND-CPA advantage is as claimed, we need a precise characterisation of the behaviour of random assignment programs, as is given by Theorem 4.3. See Smith and Alpízar (2006) for the details.

6. Related work and conclusions

This paper has shown that, under the Denning restrictions, well-typed probabilistic programs are guaranteed to satisfy an *approximate* probabilistic non-interference property, provided the probability of non-termination is small. The proof is based on a new notion of *fast simulation*, which builds on the work in Baier *et al.* (2005) on strong and weak

[†] An IND-CPA adversary $\mathscr{A}$ (Bellare and Rogaway 2005) is given an *LR oracle* of the form $\mathscr{E}_K(\mathrm{LR}(\cdot,\cdot,b))$, where K is a randomly generated key and b is an internal *selection bit*, which is either 0 or 1. When $\mathscr{A}$ sends a pair of equal-length messages (M_0, M_1) to the LR oracle, it returns $\mathscr{E}_K(M_b)$. Thus, when $\mathscr{A}$ sends a sequence of pairs of messages to the LR oracle, it either gets back encryptions of the *left* messages (if $b=0$) or encryptions of the *right* messages (if $b=1$). $\mathscr{A}$'s challenge is to guess the value of b; the encryption scheme is IND-CPA secure if an efficient $\mathscr{A}$ cannot succeed with probability significantly better than $1/2$.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

simulation on discrete and continuous Markov chains. The theorem that stripping is a fast simulation shows that the theory of probabilistic simulation can be applied fruitfully to the secure information flow problem, giving another proof technique in addition to the more common bisimulation-based approach of papers like Laud and Vene (2005), Smith and Alpízar (2006) and Fournet and Rezk (2008) on languages with cryptography, and Abadi *et al.* (1998), Smith and Volpano (1998), Smith (2003), Abadi *et al.* (2006), Focardi and Centenaro (2008) on multi-threaded languages. The recent work Alpízar and Smith (2009) on secure information flow in a distributed language also makes use of the technique of stripping and fast simulation, although in a non-probabilistic context.

Stripping is somewhat reminiscent of the work in Agat (2000), which proposes the elimination of external timing leaks in programs through a transformation-based approach. But our stripping operation is not an implementation technique, but rather a *thought experiment* that we use to better understand the behaviour of programs under the Denning restrictions.

The theme of our paper is somewhat similar to Malacaria (2007), which uses Shannon's information theory to assign a quantitative measure to the amount of leakage caused by **while** loops. However, the focus in that paper is not on leaks caused by non-termination, but rather on **while** loops that *violate* the Denning restrictions, for example, by assigning to an L variable in the body of a **while** loop with an H guard. Nevertheless, it would be interesting to explore deeper connections between that work and this.

More closely related to our work is Askarov *et al.* (2008), which explores termination-insensitive non-interference (as guaranteed by the Denning restrictions) in the context of a deterministic programming language with an **output** command. The authors observe that such programs can leak an unbounded amount of information (albeit slowly) by going into an infinite loop at some point within a sequence of outputs.

References

Abadi, M., Corin, R. and Fournet, C. (2006) Computational secrecy by typing for the pi calculus. In: Kobayashi, N. (ed.) Fourth Asian Symposium on Programming Languages and Systems (APLAS 2006). *Springer-Verlag Lecture Notes in Computer Science* **4279** 253–269.

Abadi, M., Fournet, C. and Gonthier, G. (1998) Secure implementation of channel abstractions. In: *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society 105.

Agat, J. (2000) Transforming out timing leaks. In: *Proceedings 27th ACM Symposium on Principles of Programming Languages POPL*, ACM 40–53.

Alpízar, R. and Smith, G. (2009) Secure information flow for distributed systems. In: *Proceedings Formal Aspects of Security and Trust (FAST 2009)*, Eindhoven, Netherlands.

Askarov, A., Hunt, S., Sabelfeld, A. and Sands, D. (2008) Termination-insensitive noninterference leaks more than just a bit. In: Proceedings 13th European Symposium on Research in Computer Security (ESORICS'08). *Springer-Verlag Lecture Notes in Computer Science* **5283** 333–348.

Baier, C., Katoen, J.-P., Hermanns, H. and Wolf, V. (2005) Comparative branching-time semantics for Markov chains. *Information and Computation* **200** (2) 149–214.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

Bellare, M. and Rogaway, P. (2005) Introduction to modern cryptography. Available at `http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html`.

Denning, D. and Denning, P. (1977) Certification of programs for secure information flow. *Communications of the ACM* **20** (7) 504–513.

Di Pierro, A., Hankin, C. and Wiklicky, H. (2002) Approximate non-interference. In: *Proceedings 15th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 1–17.

Feller, W. (1968) *An Introduction to Probability Theory and Its Applications*, Volume I (third edition), John Wiley and Sons.

Focardi, R. and Centenaro, M. (2008) Information flow security of multi-threaded distributed programs. In: *PLAS '08: Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ACM 113–124.

Fournet, C. and Rezk, T. (2008) Cryptographically sound implementations for typed information-flow security. In: *Proceedings 35th Symposium on Principles of Programming Languages*, ACM.

Gray, III, J. W. (1990) Probabilistic interference. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society 170–179.

Jonsson, B. and Larsen, K. (1991) Specification and refinement of probabilistic processes. In: *Proceedings 6th IEEE Symposium on Logic in Computer Science*, IEEE Computer Society 266–277.

Kemeny, J. and Snell, J. L. (1960) *Finite Markov Chains*, Van Nostrand.

Larsen, K. G. and Skou, A. (1991) Bisimulation through probabilistic testing. *Information and Computation* **94** (1) 1–28.

Laud, P. and Vene, V. (2005) A type system for computationally secure information flow. In: Proceedings of the 15th International Symposium on Fundamentals of Computational Theory. *Springer-Verlag Lecture Notes in Computer Science* **3623** 365–377.

Malacaria, P. (2007) Assessing security threats of looping constructs. In: *Proceedings 34th Symposium on Principles of Programming Languages*, ACM 225–235.

Myers, A. C., Chong, S., Nystrom, N., Zheng, L. and Zdancewic, S. (2006) Jif: Java + information flow, Cornell University. Available at `http://www.cs.cornell.edu/jif/`.

Sabelfeld, A. and Myers, A. C. (2003) Language-based information flow security. *IEEE Journal on Selected Areas in Communications* **21** (1) 5–19.

Sabelfeld, A. and Sands, D. (2000) Probabilistic noninterference for multi-threaded programs. In: *Proceedings 13th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 200–214.

Smith, G. (2003) Probabilistic noninterference through weak probabilistic bisimulation. In: *Proceedings 16th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 3–13.

Smith, G. (2006) Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security* **14** (6) 591–623.

Smith, G. and Alpízar, R. (2006) Secure information flow with random assignment and encryption. In: *Proceedings 4th ACM Workshop on Formal Methods in Security Engineering*, ACM 33–43.

Smith, G. and Alpízar, R. (2007) Fast probabilistic simulation, nontermination, and secure information flow. In *Proceedings 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ACM 67–71.

Smith, G. and Volpano, D. (1998) Secure information flow in a multi-threaded imperative language. In: *Proceedings 25th Symposium on Principles of Programming Languages*, ACM 355–364.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press

Volpano, D. and Smith, G. (1997) Eliminating covert flows with minimum typings. In: *Proceedings 10th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 156–168.

Volpano, D. and Smith, G. (1999) Probabilistic noninterference in a concurrent language. *Journal of Computer Security* **7** (2, 3) 231–253.

Volpano, D., Smith, G. and Irvine, C. (1996) A sound type system for secure flow analysis. *Journal of Computer Security* **4** (2, 3) 167–187.

https://doi.org/10.1017/S0960129511000181 Published online by Cambridge University Press